Trabajo Práctico 1
Cuatro Tree

Organización del Computador 2

Segundo Cuatrimestre 2016

1. Introducción

Este trabajo práctico consiste en implementar funciones sobre una estructura denominada *CuatroTree*. Un *CuatroTree* es un árbol de búsqueda de cuatro hijos donde cada nodo almacena tres claves.

A fin de hacer más simple la implementación, se agregarán elementos sin realizar rotaciones. Es decir, el árbol no se modificará para ser rebalanceado. Las funciones a implementar permiten crear o borrar un árbol y agregar elementos. No está permitido borrar nodos ya que implicaría rebalanceo.

Además de las funciones básicas se implementará un iterador sobre el árbol. Un iterador consiste en una estructura que almacena una posición de un elemento dentro del árbol y permite moverse al siguiente elemento de forma ordenada.

2. Tipo CuatroTree

La estructura de *CuatroTree* consiste de dos tipos diferentes de estructuras. Una para almacenar el *CuatroTree* y otra para almacenar los nodos. En la figura 1, se puede ver un ejemplo esquemático de cómo están relacionadas las diferentes estructuras.

```
#define NODESIZE 3

typedef struct ctTree_t {
  struct ctNode_t* root;
  uint32_t size;
} __attribute__((__packed__)) ctTree;

typedef struct ctNode_t {
  struct ctNode_t* father;
  uint32_t value[NODESIZE];
  uint8_t len;
  struct ctNode_t* child[NODESIZE+1];
} __attribute__((__packed__)) ctNode;
```

En la estructura `ctTree_t` se tienen dos atributos, por un lado el puntero al nodo raíz del árbol y por otro la cantidad de elementos que tiene almacenados. La estructura `ctNode_t` tiene en cambio cuatro atributos, el arreglo de elementos `value`, el arreglo de hijos `child`, la cantidad de elementos válidos dentro del arreglo `len` y el puntero al nodo padre `father`.

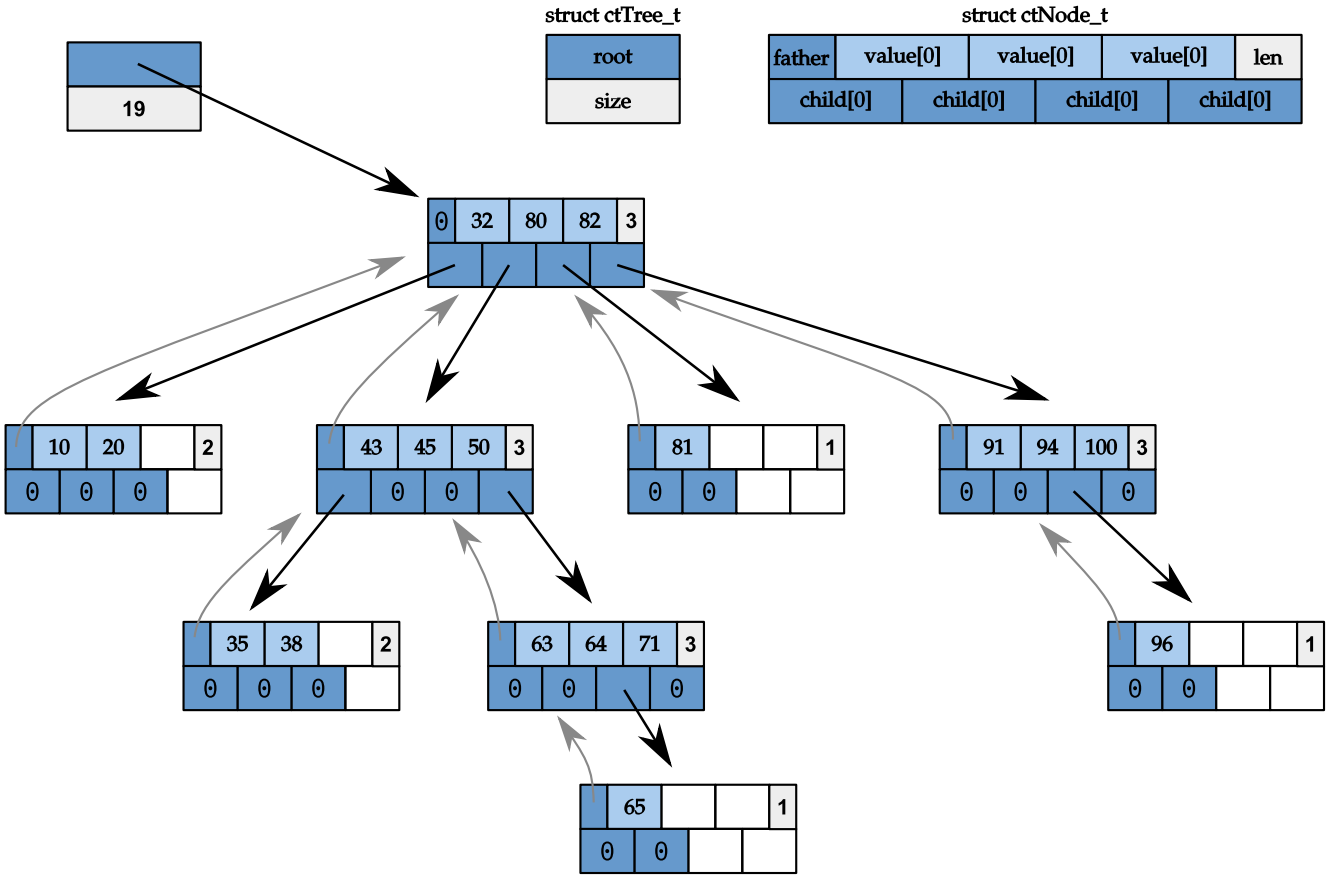

Figura 1: Ejemplo de la estructura de un *CuatroTree*

Además se considerará la siguiente estructura que será utilizada para almacenar un iterador.

```
typedef struct ctIter_t {
  ctTree* tree;
  struct ctNode_t* node;
  uint8_t current;
  uint32_t count;
} __attribute__((__packed__)) ctIter;
```

La estructura del iterador `ctIter_t` contiene la información necesaria para posicionarse en un elemento del árbol, el `nodo` en cuestión y el índice actual `current` dentro de este nodo. Además contiene un puntero al árbol que se esta recorriendo `tree` y un contador con la cantidad de elementos recorridos `count`. En la figura 2 se puede ver el funcionamiento del iterador para el ejemplo anterior.

Figura 2: Ejemplo del funcionamiento del iterador sobre el ejemplo de la figura anterior.

En el gráfico 2 se pueden reconocer los siguientes casos particulares:

- Caso 1: Al comienzo el iterador es invalido, es decir *node* es igual a cero.
- Caso 2: El estado del iterador luego de ejecutar la función que lo posiciona en el primer elemento a recorrer. Notar que el contador de elementos recorridos vale 1.
- Caso 3: Como no tiene hijos, el cursor se mueve dentro de los elementos del nodo (paso del 2 a 3), notar que se incrementa el valor de current.
- Caso 4: Se sube de nivel ya que el subárbol del nodo apuntado en el caso 3, no tiene mas elementos que recorrer. Notar que se marca en el gráfico, al subárbol en un gris oscuro, indicando ya que se recorrió.
- Caso 5: Una vez que se paso por el elemento 32 en el caso 4, se baja hasta el elemento mas chico del subárbol a la derecha del valor 32. Notar que esto implica bajar dos niveles.
- Caso 20: Este es el iterador que apunta al ultimo elemento válido. El siguiente paso será determinar que ya se recorrieron todos los elementos del árbol y se pasará

Funciones de CuatroTree

- `void ct_new(ctTree** pct);`
  Crea un árbol vacío.
- `void ct_delete(ctTree** pct);`
  Borra un árbol con todos sus nodos.
- `void ct_add(ctTree* ct, uint32_t value);`
  Agrega un elemento al árbol de forma ordenada.
- `void ct_print(ctTree* ct, FILE *pFile);`
  Imprime en *pFile*, los elementos de un árbol de forma ordenada de a uno por línea.

Funciones de Iterador sobre CuatroTree

- `ctIter* ctIter_new(ctTree* ct);`
  Crea un nuevo iterador e inicializa su estructura. Inicialmente es un interador inválido, es decir `node=0`.
- `void ctIter_delete(ctIter* ctIt);`
  Borra un iterador.
- `void ctIter_first(ctIter* ctIt);`
  Posiciona el iterador en el primer elemento a recorrer. Es decir, busca en el árbol el elemento más pequeño.
- `void ctIter_next(ctIter* ctIt);`
  Posiciona el iterador en el siguiente elemento del árbol.
- `uint32_t ctIter_get(ctIter* ctIt);`
  Obtiene el elemento apuntado por el iterador.
- `uint32_t ctIter_valid(ctIter* ctIt);`
  Determina si el iterador es válido. Es decir, que `node` es distinto de *null*.

Detalle sobre funciones

A fin de simplificar la implementación del trabajo, se explica a continuación el funcionamiento de dos de las funciones clave del TP. La función encargada de agregar elementos dentro del árbol y la función que mueve el iterador al próximo elemento.

Detalle sobre función `ct_add`

Para resolver esta función se deben resolver dos pasos. El primero consiste en encontrar el nodo donde se debe agregar el nuevo elemento y el segundo en agregar el elemento al nodo.

```
ct_add(tree, newValue):

    node = search(tree.root, newValue)
    fill(node, newVal)
    tree.size = tree.size + 1
```

Se recomienda construir las siguientes funciones:

- `ctNode* ct_aux_search(ctNode** currNode, ctNode* fatherNode, uint32_t newVal);`
  Busca recursivamente un nodo donde agregar el elemento de forma ordenada. En el caso de encontrar un nodo con espacio, retorna dicho nodo. De no existir tal nodo, genera un nuevo nodo, lo conecta al árbol y retorna el nodo en cuestión.

- `void ct_aux_fill(ctNode* currNode, uint32_t newVal);`
  Dado un nodo busca dónde agregar ordenadamente el nuevo elemento. Mueve los elementos del vector de valores y hijos según corresponda para hacer el espacio necesario. Por último incremeta la cantidad de elementos del nodo.

Detalle sobre función `ctIter_next`

La función `next` del iterador se resuelve en dos casos bien definidos. En uno de ellos me debo mover hacia las hojas del árbol y en el otro hacia la raíz. La función determinará cuál caso es y llamará a una función que resuelve cada uno de forma recursiva.

```
ctIter_next(ctIt):

    // Incremento al siguiente elemento
    ctIt.current = ctIt.current + 1

    // Si el hijo no existe, entones sigo en el mismo nodo
    if ctIt.node.child[ctIt.current] == 0 :

        // si recorri todos los elementos del nodo subo
        if ctIt.current > ctIt.node.len-1 :
            ctIter_aux_up(ctIt)

        // salgo, ya movi el iterador y no hay hijos
        return

    // hay hijos
    else

        // asigno un nuevo nodo en el iterador
        ctIt.node = ctIt.node.child[ctIt.current]
        // bajo hasta encontrar el menor del subarbol
        ctIter_aux_down(ctIt);
```

Se recomienda construir las siguientes funciones:

- `void ctIter_aux_down(ctIter* ctIt)`
  Dado un iterador, lo mueve hasta el elemento más pequeño del subárbol construido por el nodo apuntado por el iterador.

- `void ctIter_aux_up(ctIter* ctIt)`
  Mueve el iterador al padre del nodo actual hasta encontrar un nodo donde falten recorrer elementos. Para esto utiliza la función a continuación, que indica qué índice dentro del vector de hijos es el nodo actual.

- `uint32_t ctIter_aux_isIn(ctNode* current, ctNode* father)`
  Determina el índice dentro del vector de hijos para *current* con respecto a *father*.

Tener en cuenta que esta función además debe invalidar al iterador una vez que este recorrió la totalidad de los elementos del árbol. Este caso no esta descripto en el seudocodigo anterior.

3. Enunciado

Ejercicio 1

Implementar todas las funciones de **CuatroTree** mencionadas anteriormente según se indica a continuación:

En lenguaje assembler

- `void ct_new(ctTree** pct);` *(35 líneas)*
- `void ct_delete(ctTree** pct);` *(33 líneas)*
- `void ct_print(ctTree* ct, FILE *pFile);` *(26 líneas)*
- `ctIter* ctIter_new(ctTree* ct);` *(9 líneas)*
- `void ctIter_delete(ctIter* ctIt);` *(1 líneas)*
- `void ctIter_first(ctIter* ctIt);` *(9+9 líneas)*
- `void ctIter_next(ctIter* ctIt);` *(58 líneas)*
- `uint32_t ctIter_get(ctIter* ctIt);` *(4 líneas)*
- `uint32_t ctIter_valid(ctIter* ctIt);` *(4 líneas)*

En lenguaje C

- `void ct_add(ctTree* ct, uint32_t value);` *(60 líneas)*

Entre paréntesis se menciona de forma ilustrativa, la cantidad de líneas que tomo cada función en la solución de la cátedra.

Ejercicio 2

Construir un programa de prueba (modificando `main.c`) que realice las siguientes acciones llamando a las funciones implementadas anteriormente:

1- Crear un árbol nuevo

2- Agregar los siguientes valores uno a uno:
`10`, `50`, `30`, `5`, `20`, `40`, `60`, `19`, `39`, `4`

3- Crear un Iterador

4- Usando el iterador, recorrer el árbol e imprimir todos los elementos

5- Destruir el iterador

6- Destruir el árbol

Testing

En un ataque de bondad, hemos decidido entregarle una serie de *tests* o pruebas para que usted mismo pueda verificar el buen funcionamiento de su código.

Luego de compilar, puede ejecutar `./tester.sh` y eso probará su código. El test realizará una serie de operaciones, estas generarán archivos que serán comparados las soluciones provistas por cátedra. Además, el test realizará pruebas sobre la correcta administración de la memoria dinámica.

Archivos

Se entregan los siguientes archivos:

- `cuatrotree.asm`: Archivo a completar con su código assembler.
- `cuatrotree.c`: Archivo a completar con su código C.
- `main.c`: Archivo para completar la solución del ejercicio 2.
- `Makefile`: Contiene las instrucciones para compilar `tester` y `main`. No debe modificarlo.
- `cuatrotree.h`: Contiene la definición de la estructura y funciones. No debe modificarlo.
- `tester.c`: Codigo de testing. No debe modificarlo.
- `tester.sh`: Script que realiza todos los test. No debe modificarlo.
- `Catedra.salida.caso.chico.txt` y `Catedra.salida.caso.grande.txt`: Resultados de la cátedra. No deben modificarlos.

Notas:

a) Para todas las funciones hechas en lenguaje ensamblador que llamen a cualquier función extra, ésta también debe estar hecha en lenguaje ensamblador. (Idem para código C).

b) Toda la memoria dinámica reservada por la función `malloc` debe ser correctamente liberada, utilizando la función `free`.

c) Para el manejo de archivos se recomienda usar las funciones de C: `fopen`, `fread`, `fwrite`, `fclose`, `fseek`, `ftell`, `fprintf`, etc.

d) Para poder correr los test, se debe tener instalado *Valgrind* (En ubuntu: `sudo apt-get install valgrind`).

e) Para corregir los TPs usaremos los mismos tests que les fueron entregados. El criterio de aprobación es que el TP supere correctamente todos los tests y no contenga errores de forma.

4. Informe y forma de entrega

Para este trabajo práctico no deberán entregar un informe. En cambio, deberán entregar un archivo comprimido con el mismo contenido que el dado para realizarlo, habiendo modificado los archivos `cuatrotree.asm`, `cuatrotree.c` y `main.c`.

La fecha de entrega de este trabajo es Martes 06/09. Deberá ser entregado a través de la página web. El sistema sólo aceptará entregas de trabajos hasta las 17:00 hs del día de entrega.
Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes
`orga2-doc@dc.uba.ar`.

Aclaración importante: No incluir archivos binarios u objetos resultantes de la compilación o ensamblado del TP.